

Cache_Lab

20220124 김문경

Part A

Part A는 캐시의 기능을 수행하는 시뮬레이터 코드를 짜는 것이다. 이때 해당 시뮬레이터의 Replacement Policy는 LRU(Least Recently Used) 방식을 채택한다는 조건이 존재한다.

Cache Simulator Process

1. Parsing
2. Cache Init
3. Cache Simulator
4. I/O

먼저 캐시의 hit 횟수와 miss 횟수, eviction 횟수를 저장할 변수를 선언한다.

```
int hit_count = 0;
int miss_count = 0;
int eviction_count = 0;
```

그 다음 캐시 시뮬레이터의 전신이 되는 캐시를 배열을 통해 구현하였다.

```
typedef struct{
    bool valid;
    int tag;
    int age;
} line;

typedef line* set;

typedef set* caches;

caches cache;
```

valid bit을 의미하는 bool형 변수 valid, tag bit를 의미하는 int형 변수 tag, 그리고 Replacement Policy인 LRU를 적용하기 위한 int형 변수 age를 가진 구조체 line을 선언하였다. 그리고 여러 개의 line들이 모여서 만들어지는 set, 그리고 그 여러 개의 set들이 모인 caches라는 데이터형을 선언했다. 그리고 caches형 변수 cache를 선언하여, 캐시 시뮬레이터의 '캐시' 역할을 하는 배열을 만들었다.

이때 age는

```
int LRU = 0;
```

LRU라는 변수를 통해서 갱신시키는 메커니즘을 적용하였다. 캐시에 데이터가 들어올 때마다 LRU를 증가시키고, 들어온 데이터의 line의 age에 증가된 LRU를 저장시켜, 최근에 들어온 데이터일수록 높은 age를, 반대로 evict되지 않고 오래 캐시에 남아있는 데이터일수록 낮은 age를 가지는 구조가 되도록 했다.

```
int s;
int E;
int b;
char* t;
```

그 다음으로 command를 통해 들어오는 set 개수를 의미하는 s, line 개수를 의미하는 E, block offset을 의미하는 b를 받아들인다. 그리고 trace 파일 이름을 저장할 문자열 변수 t를 선언한다.

```
bool display = false;
```

또한 -v command 실행 시 hit, miss, eviction의 로그를 모두 출력해야 하므로 해당 기능을 수행하기 위해 -v command 여부를 저장하는 변수 display를 선언한다.

```
while((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1){  
    switch(opt){  
        case 'h':  
            printf("Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>");  
            break;  
        case 'v':  
            display = true;  
            break;  
        case 's':  
            s = atoi(optarg);  
            break;  
        case 'E':  
            E = atoi(optarg);  
            break;  
        case 'b':  
            b = atoi(optarg);  
            break;  
        case 't':  
            t = optarg;  
            break;  
    }  
}  
}
```

가장 처음으로 command를 통해 들어오는 -v, s, E, b, t에 대한 정보들을 getopt를 통해 저장한다. 이때 command는 문자열로 들어오므로 int형 변수의 경우 atoi를 통해서 변환하여 저장한다.

```
cache = (set*)malloc(sizeof(set) * (1<<s));  
  
for(int i = 0; i < (1<<s); i++){  
  
    cache[i] = (line*)malloc(sizeof(line) * (1<<E));  
  
    for(int j=0;j<(1<<E);j++){  
  
        cache[i][j].valid = false;  
        cache[i][j].tag = 0;  
        cache[i][j].age = 0;  
  
    }  
}
```

저장한 s, E, b에 대한 정보를 바탕으로 주어진 크기만큼 캐시 배열을 동적할당한다. 이와 동시에 할당한 cache 배열 원소들에 대한 valid, tag, age를 initialization한다.

```

trace = fopen(t,"r");

while(fscanf(trace, " %c %llx, %d", &operation, &address, &size) != EOF){

    tag = address >> (s+b);
    set_index = (address >> b) & ((1<<s) - 1);

    if(display == true){
        printf("%c %llx, %d ", operation, address, size);
    }

    switch(operation){
        case 'L':
        case 'S':
            if(hit_miss(tag, set_index) == -1){
                eviction(tag, set_index);
            }
            break;

        case 'M':
            for(int i=0;i<2;i++){
                if(hit_miss(tag, set_index) == -1){
                    eviction(tag, set_index);
                }
            }
            break;
    }

    printf("\n");
}

```

이제 본격적으로 trace파일에 따른 캐시의 hit, miss, eviction이 일어나는 과정을 구현할 차례이다. 먼저 trace파일을 연다. 이대 trace파일의 모든 행들은 "[operation] address size" 형태로 통일되어 있기 때문에 각각의 줄을 해당 형태로 읽어들인다.

읽어들인 address에서 hit, miss, eviction에 영향을 주는 tag 값과 set index 값을 추출한다. 이때 tag, set_index라는 새로운 변수를 이용해서 해당 값을 추출한다. 이 과정에서 address의 구조가 [tag - set index - block offset]이라는 것을 이용하여 tag와 set index를 얻어낸다.

이후 operation에 따라 다른 방식으로 캐시에 접근한다. operation의 경우 'L', 'S', 'M'이 있는데, L은 load, S는 store, M은 modify를 의미한다. 이때 modfiy는 load와 store 모두 진행하는 것으로 생각해도 된다. 이에 따라 load, store은 캐시에 1번 접근하고, modify는 2번 접근하는 형태의 코드를 작성했다.

이때 캐시에 접근하는 프로세스를 의미하는 hit_miss함수를 만들어 이를 적용했다. hit_miss함수는 다음과 같다.

```
int hit_miss(unsigned long long int tag, unsigned long long int set_index){

    /*hit process*/
    for(int i=0;i<E;i++){
        if(cache[set_index][i].tag == tag){ //check if the tag matches
            if(cache[set_index][i].valid == true){ //check the valid bit

                hit_count++;
                if(display == true){
                    printf("hit ");
                }

                LRU++;
                cache[set_index][i].age = LRU;

                return 1; //if hit, quit function
            }
        }
    }

    /*miss process*/
    miss_count++;

    if(display == true){
        printf("miss ");
    }

    return -1;
}
```

캐시 접근 프로세스 순서는 수업 ppt를 참조했다. 참조하려는 데이터 address의 tag와 같은 tag를 가진 캐시 데이터가 있는지 확인하고, 있다면 해당 데이터가 유효한 값인지 valid를 확인한다. valid가 true라면 hit가 발생한 것이므로 hit_count를 1 증가시키고, 새롭게 hit된 데이터의 age를 갱신한다. 갱신 과정은 다음과 같다: LRU를 증가시킨 후, 그리고 증가된 LRU를 age에 저장한다. 그리고 난 후 1을 반환하여 함수를 종료한다.

만약 모든 line에 접근했음에도 불구하고 함수가 끝나지 않았다면, hit가 발생하지 않았다는 것을 의미한다. 즉, miss가 발생했다는 것이다. miss가 발생했으므로 miss count를 증가시키고, -1을 반환하여 함수를 종료시킨다.

이때 miss가 발생했으므로 새롭게 캐시가 교체되어야 한다. 그래서 hit_miss 함수가 -1을 반환할 경우, 캐시 교체 프로세스를 진행하는 eviction 함수를 실행시킨다.

```
if(hit_miss(tag, set_index) == -1){
    eviction(tag, set_index);
}
break;
```

eviction 함수는 다음과 같다.

```
void eviction(unsigned long long int tag, unsigned long long int set_index){

    unsigned long long int oldst = cache[set_index][0].age;
    int change = 0;

    for(int i=0;i<E;i++){ // oldest block finding
        if(cache[set_index][i].age < oldst){
            oldst = cache[set_index][i].age; // renewal oldest block
        }
    }

    for(int i=0;i<E;i++){ // find line which have to evict
        if(cache[set_index][i].age == oldst){
            change = i;
        }
    }

    if(cache[set_index][change].valid == true){

        eviction_count++;
        if(display == true){
            printf("eviction ");
        }
    }

    /*replacing value*/
    cache[set_index][change].valid = true;
    cache[set_index][change].tag = tag;
    cache[set_index][change].age = LRU;

    return;
}
```

eviction의 경우 새롭게 데이터가 기존 데이터를 대체하는 것이기 때문에, replacement policy가 적용된다. replacement policy인 LRU에 따라 캐시 내에서 가장 오래된 데이터가 evict되어 하므로, set 내 데이터들의 age값을 비교하면서 가장 작은 age, 즉 가장 오래된 데이터를 찾는다. 해당 데이터의 age를 저장하는 oldst 변수, 그리고 가장 오래된 데이터의 순번을 저장하는 change 변수를 선언한다. 그리고 난 후 해당 순번에 존재하는 valid, tag, age를 새로운 데이터로 변경한다.

```

for(int i=0;i<(1<<s);i++){ // deallocating
    free(cache[i]);
}
free(cache);

fclose(trace); // file close

printSummary(hit_count, miss_count, eviction_count);

return 0;

```

이후 trace파일을 모두 읽어들였으면 캐시 시뮬레이터가 끝난 것으로, 사용했던 캐시 배열을 할당해제하고, trace파일을 닫는다. 그리고 난 후 기록된 hit, miss, eviction 카운트를 printSummary를 통해 출력한 후 함수를 종료한다.

Part B

/Part B의 경우 진행 과정에서 서버 접속이 되지 않는 문제가 발생하여, Local 환경에서 WSL을 이용해 진행했음을 알린다./

Part B는 주어진 Matrix를 Transpose하는 과정에서 발생하는 miss 횟수를 최소화하는 문제이다. 해당 문제를 해결하기 위해, 수업 시간에 배운 Blocking Method를 활용해보기로 했다.

1. 32X32

Transpose function을 작성하기 전, 32X32 Matrix에 대하여 Locality를 고려하지 않는 단순 Transpose 함수의 miss 횟수를 확인해보았다.

```
func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152
```

miss 횟수가 1184회로, 점수 획득 제한인 300회보다 훨씬 많은 miss를 발생시켰다.

또한 변환하고자 하는 A Matrix의 모든 원소들에 대한 set index를 확인해보았다.

```

void addr_print(int A[256][256]){
    for(int i=0;i<32;i++){
        for(int j=0;j<32;j++){

            int set_index = (((long)(&A[0][0])+32*i+j) >> 5) & 31;
            printf("%4d",set_index);

        }
        printf("\n");
    }
}

```

tracegen.c 코드에 위의 코드를 삽입하여 출력하는 형태로 확인하였다.

A Matrix의 경우 아래와 같다.

B Matrix의 경우 아래와 같다.

이를 통해 같은 열의 원소들은 set index가 모두 같다는 점, 같은 열의 경우 Matrix 종류가 달라도 set index가 같다는 점, 모든 열의 set index가 다르다는 점을 파악할 수 있었다.

다음으로는 tag를 확인해보았다. 확인을 위해 사용한 코드는 아래와 같다.

```
void tag_print(int A[256][256], int B[256][256]){
    for(int i=0;i<32;i++){
        int set_index = (((long)(&A[0][0]))+32*i) >> 10;
        int set_index2 = (((long)(&B[0][0]))+32*i) >> 10;
        printf("%x %x\n",set_index,set_index2);

    }
    printf("\n");
}
```

결과는 아래와 같다

행이 같은 경우 set index는 같았지만, 위 그림에서 보이듯 tag의 값은 다음을 알 수 있었다.

본격적으로 32X32 Matrix Transpose function을 작성해보았다. int 크기가 4이고, Block offset의 크기가 5bit이므로 Block의 크기는 $(2^5) / 4 = 8$ 이 적절하다고 판단하여 Block의 크기를 8로 정했다. 이후 blocking method를 사용하여 optimization을 진행했다.

```
if (M == 32 && N == 32)
{
    for (int i = 0; i < 32; i += 8)
    {
        for (int j = 0; j < 32; j += 8)
        {
            for (int il = i; il < i + 8; il++)
            {
                for (int jl = j; jl < j + 8; jl++)
                {
                    B[jl][il] = A[il][jl];
                }
            }
        }
    }
}
```

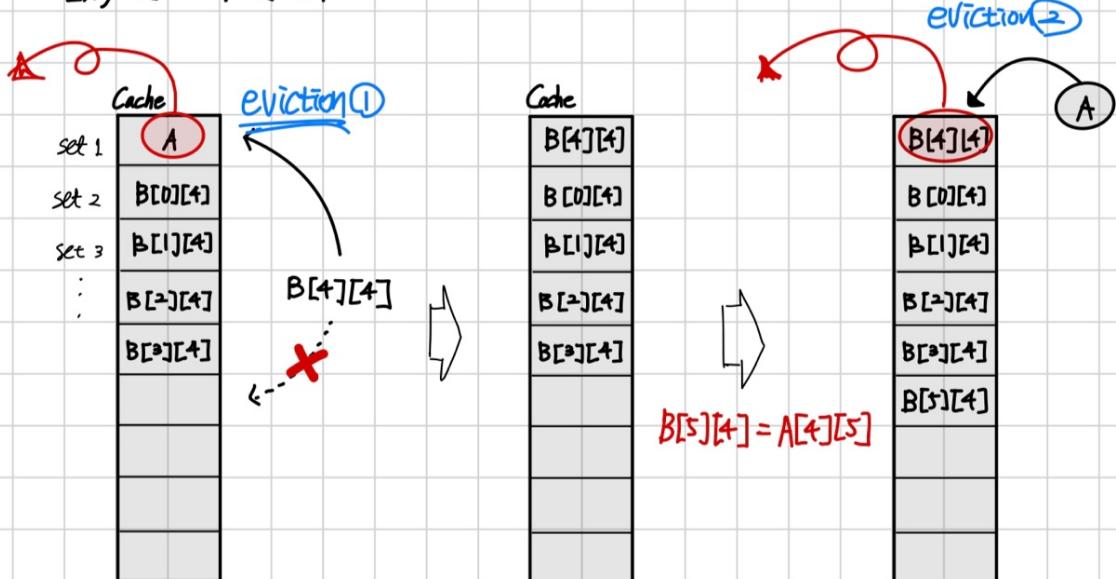
그러나 해당 코드를 테스트한 결과,

func 0 (Transpose submission): hits:1709, misses:344, evictions:312

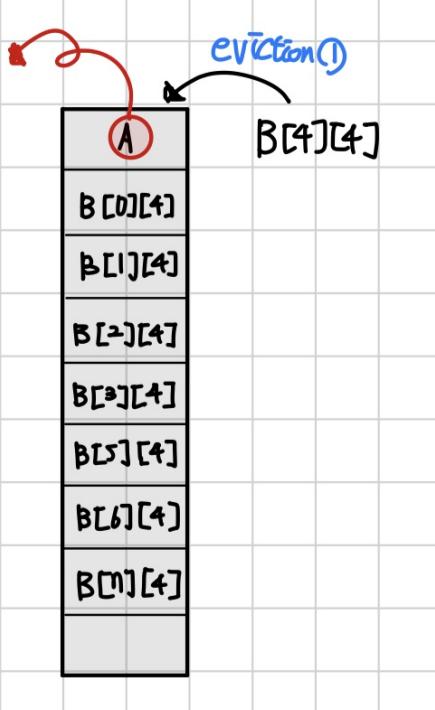
miss 횟수가 344회가 나왔다. 초기 1184회보다는 획기적으로 많이 줄어들었으나, 아직 300회 미만의 조건을 충족하지 못했다. Blocking만으로는 부족하다는 것을 알 수 있었다.

추가적으로 miss 횟수를 줄일 수 있는 방안을 생각하던 중, 행렬의 대각선 원소를 이용하면 좋을 것 같다는 생각이 들었다. 대각선 원소들은 A와 B의 행이 같으므로, 앞서 address 확인을 통해 알아본 바에 따라, 대각선 원소들에 대해서는 A의 원소와 B의 원소의 set index가 같다. 그런데 tag bit은 다르고, E = 1이기 때문에, 한 set당 line이 한 개이기 때문에 위의 경우 불필요한 eviction이 발생하게 된다.

Ex) $il = 4$ 인 경우



위의 그림을 예시로 보면, A행과 B[4][4]의 set index가 같기 때문에 B[4][4]를 참조할 때 기존 캐시에 저장되어 있던 A행 데이터가 evict된다. 그 다음 loop에서 진행되는 B[5][4]의 데이터에 A[4][5]의 데이터를 넣는 과정에서, A행의 데이터가 다시 필요하기 때문에 B[4][4] 데이터가 evict된다. 하지만 이렇게 두 번의 eviction이 발생하는 것은 불필요하다고 생각이 들었다.



B[4][4]의 참조를 제일 뒤로 미룬 후, A[4][4]의 데이터를 임의의 변수에 저장한 뒤 B[0][4]부터 B[7][4]의 참조가 모두 끝난 후 B[4][4]를 참조하는 형태의 프로세스는 eviction이 한 번만 발생하여 불필요한 miss를 줄일 수 있다. B의 원소들을 참조할 때마다 A 데이터가 무조건 참조되기 때문에 B 원소들의 참조가 끝날 때까지는 A가 캐시에 항상 존재해야 한다. 따라서 캐시에 존재하는 A 데이터를 evict하지 않고도 캐싱할 수 있는 다른 B의 원소들을 모두 참조한 뒤, A가 더 이상 캐시에 존재하지 않아도 되는 상황에서 B[4][4]를 참조하여 A를 evict하게 되면 전체 eviction이 한번만 일어나도 모든 데이터를 참조할 수 있게 되는 것이다.

그러한 프로세스를 적용한 코드는 다음과 같다.

```
if (M == 32 && N == 32)
{
    int temp;

    for (int i = 0; i < 32; i += 8)
    {
        for (int j = 0; j < 32; j += 8)
        {
            for (int il = i; il < i + 8; il++)
            {
                for (int jl = j; jl < j + 8; jl++)
                {
                    if (jl == il)
                    {
                        temp = A[il][jl];
                    }
                    if (jl != il)
                    {
                        B[jl][il] = A[il][jl];
                    }
                }
                if (i == j)
                {
                    B[il][il] = temp;
                }
            }
        }
    }
}
```

j과 il이 같은 경우, 즉 대각선 원소의 경우 temp라는 변수에 저장해 둔 후, jl에 대한 loop가 끝난 이후에 B[il][jl]에 저장하는 방식이다. jl과 il이 다른 경우 기존 방식대로 저장한다.

2. 64X64

문제를 해결하기 위한 첫 번째 시도로 64X64 배열에 32X32 의 transpose를 적용해보았다. 그래서 32X32 Matrix 4개로 나눈 후 각각에 대하여 32X32 버전 transpose 코드를 적용시켰는데, 이전과 달리 획기적으로 줄어들지 않았다.

단순 transpose 코드의 경우

```
func 1 (Simple row-wise scan transpose): hits:3473, misses:4724, evictions:4692
```

miss횟수가 4724회인 반면, 해당 코드의 경우

```
func 0 (Transpose submission): hits:3561, misses:4636, evictions:4604
```

로 많이 줄지 않았다.

이는 32X32에 비해 늘어난 크기로 인한 eviction case의 증가와, 캐시의 set 수가 32개인 것 때문으로 보인다.

64X64 Matrix에 대한 set index를 확인해봤다. 64X64의 경우 크기가 매우 커서 txt파일에 저장하는 코드를 추가하여 확인했다.

set index를 보면, 이전과 달리 하나의 행에 두 가지의 set index가 존재하는 것을 확인할 수 있다. 또한 0부터 31까지의 set index가 한번씩만 나왔던 이전과 달리 같은 set index를 가진 곳들이 많이 보인다. 이는 또 다른 miss를 발생시키는데,

위 그림과 같이 A에서 B로 transpose시킬 때 set index가 겹치는 case가 발생한다. 이럴 경우 추가적인 eviction이 발생한

다. 전체 64X64 index 중에서 이러한 case는 총 4쌍이 존재한다.

set index가 겹치는 case들에서 나타나는 추가적인 eviction을 해결하기 위해, 8x8 사이즈의 블럭을 위, 아래로 8x4 블럭 두 개로 나누어 transpose를 실행했다.

	0	1	2	3	4	5	6	7
a1	a1	
b1	b1	
c1	-	-	-	-	-	-	c1	
d1	d1	

우선 A 행렬의 각 행을 a, b, c, \dots, e, f, g 로 명명하겠다. 그리고 각 원소의 이름을 a_1, b_3 과 같이 명명하겠다. 가장 처음으로 위 4행인 $a \sim d$ 에 대해 B로 transpose한다. 이때 아래쪽 4칸의 열에는 원소를 넣지 않고 오른쪽 편에 있는 행에 4개의 원소를 차곡차곡 넣어 그림과 같이 임시로 저장해둔다.

	0	1	2	3	4	5	6	7
0	a1	b1	c1	d1	a5	b5	c5	d5
1	a2	b2	c2	d2	a6	b6	c6	d6
2	a3	b3	c3	d3	a7	b7	c7	d7
3	a4	b4	c4	d4	a8	b8	c8	d8
4								
5								
6								
7								

	0	1	2	3	4	5	6	7
0	a1	b1	c1	d1				
1	a2	b2	c2	d2				
2	a3	b3	c3	d3				
3	a4	b4	c4	d4				
4	a5	b5	c5	d5				
5	a6	b6	c6	d6				
6	a7	b7	c7	d7				
7	a8	b8	c8	d8				

그 다음 transpose를 완성하기 위해 오른쪽 위쪽 4x4 칸에 저장해두었던 원소들을 원래대로 배치시킨다.

이후 A의 아래쪽 8x4 데이터들을 B로 옮긴다.

<A>

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4	e1	e2	e3	e4				
5	f1	f2	f3	f4				
6	g1	g2	g3	g4				
7	h1	h2	h3	h4				

	0	1	2	3	4	5	6	7
0	e1	f1	g1	h1				
1	e2	f2	g2	h2				
2	e3	f3	g3	h3				
3	e4	f4	g4	h4				
4								
5								
6								
7								

불필요한 eviction을 방지하기 위해, 우선 아래쪽 왼쪽 4x4 데이터들을 transpose한다.

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4		e5 f5 g5 h5						
5		c6 d6 e6 h6						
6		e7 f7 g7 h7						
7		e8 f8 g8 h8						

이후 오른쪽 아래 transpose를 완료하였다.

이와 같은 프로세스를 적용한 코드는 아래와 같다.

```

if (M == 64 && N == 64)
{
    int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    for (int i = 0; i < 64; i += 8)
    {
        for (int j = 0; j < 64; j += 8)
        {
            for (int il = i; il < i + 4; il++)
            {
                tmp0 = A[il][j];
                tmp1 = A[il][j + 1];
                tmp2 = A[il][j + 2];
                tmp3 = A[il][j + 3];
                tmp4 = A[il][j + 4];
                tmp5 = A[il][j + 5];
                tmp6 = A[il][j + 6];
                tmp7 = A[il][j + 7];

                B[j][il] = tmp0;
                B[j + 1][il] = tmp1;
                B[j + 2][il] = tmp2;
                B[j + 3][il] = tmp3;
                B[j][il + 4] = tmp4;
                B[j + 1][il + 4] = tmp5;
                B[j + 2][il + 4] = tmp6;
                B[j + 3][il + 4] = tmp7;
            }

            for (int jl = j; jl < j + 4; jl++)
            {
                tmp0 = B[jl][i + 4];
                tmp1 = B[jl][i + 5];
                tmp2 = B[jl][i + 6];
                tmp3 = B[jl][i + 7];

                B[jl + 4][i] = tmp0;
                B[jl + 4][i + 1] = tmp1;
                B[jl + 4][i + 2] = tmp2;
                B[jl + 4][i + 3] = tmp3;
            }

            for (int jl = j; jl < j + 4; jl++)
            {
                tmp0 = A[i + 4][jl];
                tmp1 = A[i + 5][jl];
                tmp2 = A[i + 6][jl];
                tmp3 = A[i + 7][jl];

                B[jl][i + 4] = tmp0;
                B[jl][i + 5] = tmp1;
                B[jl][i + 6] = tmp2;
                B[jl][i + 7] = tmp3;
            }

            for (int il = i; il < i + 4; il++)
            {
                tmp0 = A[il + 4][j + 4];
                tmp1 = A[il + 4][j + 5];
            }
        }
    }
}

```

```

        tmp2 = A[il + 4][j + 6];
        tmp3 = A[il + 4][j + 7];

        B[j + 4][il + 4] = tmp0;
        B[j + 5][il + 4] = tmp1;
        B[j + 6][il + 4] = tmp2;
        B[j + 7][il + 4] = tmp3;
    }
}
}
```

그런데 본 코드의 miss 값을 검사해보니,

func 0 (Transpose submission): hits:8609, misses:1636, evictions:1604

만점 기준인 1300회보다는 더 많은 1636회가 나왔다. 따라서 해당 코드의 miss 횟수를 더 줄일 수 있는 방법에 대해 생각을 더 해보기로 했다.

```

for(int jl = j; jl < j + 4; jl++){
    tmp0 = B[jl][i + 4];
    tmp1 = B[jl][i + 5];
    tmp2 = B[jl][i + 6];
    tmp3 = B[jl][i + 7];

    B[jl + 4][i] = tmp0;
    B[jl + 4][i + 1] = tmp1;
    B[jl + 4][i + 2] = tmp2;
    B[jl + 4][i + 3] = tmp3;
}

for (int jl = j; jl < j+ 4; jl++)
{
    tmp0 = A[i + 4][jl];
    tmp1 = A[i + 5][jl];
    tmp2 = A[i + 6][jl];
    tmp3 = A[i + 7][jl];

    B[jl][i + 4] = tmp0;
    B[jl][i + 5] = tmp1;
    B[jl][i + 6] = tmp2;
    B[jl][i + 7] = tmp3;
}

```

우선 작성한 코드에서 중복되는 반복문이 있어, 변수를 변경한 후 반복문을 합쳐주었다.

```

for(int jl = j; jl < j + 4; jl++){
    tmp0 = B[jl][i + 4];
    tmp1 = B[jl][i + 5];
    tmp2 = B[jl][i + 6];
    tmp3 = B[jl][i + 7];

    B[jl + 4][i] = tmp0;
    B[jl + 4][i + 1] = tmp1;
    B[jl + 4][i + 2] = tmp2;
    B[jl + 4][i + 3] = tmp3;

    tmp4 = A[i + 4][jl];
    tmp5 = A[i + 5][jl];
    tmp6 = A[i + 6][jl];
    tmp7 = A[i + 7][jl];

    B[jl][i + 4] = tmp4;
    B[jl][i + 5] = tmp5;
    B[jl][i + 6] = tmp6;
    B[jl][i + 7] = tmp7;
}

```

그런 후 miss 횟수를 확인해본 결과,

```
func 0 (Transpose submission): hits:8561, misses:1684, evictions:1652
```

miss 횟수가 더 늘어났다. 해당 코드의 locality를 높이면 miss 횟수를 줄일 수 있을 것이라 판단했다. 위 코드에서는 B Matrix에 대한 참조와 A Matrix에 대한 참조가 번갈아 나타나는데,

```

for(int jl = j; jl < j + 4; jl++){
    tmp4 = A[i + 4][jl];
    tmp5 = A[i + 5][jl];
    tmp6 = A[i + 6][jl];
    tmp7 = A[i + 7][jl];

    tmp0 = B[jl][i + 4];
    tmp1 = B[jl][i + 5];
    tmp2 = B[jl][i + 6];
    tmp3 = B[jl][i + 7];

    B[jl + 4][i] = tmp0;
    B[jl + 4][i + 1] = tmp1;
    B[jl + 4][i + 2] = tmp2;
    B[jl + 4][i + 3] = tmp3;

    B[jl][i + 4] = tmp4;
    B[jl][i + 5] = tmp5;
    B[jl][i + 6] = tmp6;
    B[jl][i + 7] = tmp7;
}

```

아래 코드로 변경하여 번갈아서 나타나는 구조를 바꾸어 A의 참조가 끝난 후에는 오로지 B의 참조만 이루어지도록 하여 locality를 높였다. 그 결과

```
func 0 (Transpose submission): hits:8577, misses:1668, evictions:1636
```

1668회로 이전에 비해서는 줄어들었으나, 아직 1300회 미만의 조건을 충족하지 못했다. 추가적으로 locality를 높이는

방안을 생각해보았는데, B[jl] 행끼리 코드 상에서 근처에 있도록 변경하면 위치적으로 지역성이 높아지면서 중간에 참조되는 다른 행들에 대한 참조가 줄어들어 불필요한 miss가 감소할 것이라고 판단했다.

```
for(int jl = j; jl < j + 4; jl++){
    tmp4 = A[i + 4][jl];
    tmp5 = A[i + 5][jl];
    tmp6 = A[i + 6][jl];
    tmp7 = A[i + 7][jl];

    tmp0 = B[jl][i + 4];
    tmp1 = B[jl][i + 5];
    tmp2 = B[jl][i + 6];
    tmp3 = B[jl][i + 7];

    B[jl][i + 4] = tmp4;
    B[jl][i + 5] = tmp5;
    B[jl][i + 6] = tmp6;
    B[jl][i + 7] = tmp7;

    B[jl + 4][i] = tmp0;
    B[jl + 4][i + 1] = tmp1;
    B[jl + 4][i + 2] = tmp2;
    B[jl + 4][i + 3] = tmp3;
}
```

그러한 생각에 착안하여 위와 같은 코드로 변경했고, 해당 코드의 miss 횟수를 검사한 결과

```
func 0 (Transpose submission): hits:9065, misses:1180, evictions:1148
```

1180회로, 1300회 이하의 miss를 만들어낼 수 있었다.

61x67

64x64 와 마찬가지로 처음에는 32x32의 transpose 코드를 적용해보았다. 하지만 이번 Matrix의 경우 형태가 square이 아니기 때문에, transpose가 불필요한 부분에 대해서 각각 il, jl for문에 il < 67, jl < 61이라는 제한을 걸어 오로지 61x67 배열에서만 transpose가 진행되도록 했다.

```

if (M == 61 && N == 67)
{
    int temp;

    for (int i = 0; i < 67; i += 8)
    {
        for (int j = 0; j < 61; j += 8)
        {
            for (int il = i; il < 67 && il < i + 8; il++)
            {
                for (int jl = j; jl < 61 && jl < j + 8; jl++)
                {
                    if (jl == il)
                    {
                        temp = A[il][jl];
                    }
                    if (jl != il)
                    {
                        B[jl][il] = A[il][jl];
                    }
                }
                if (i == j)
                {
                    B[il][il] = temp;
                }
            }
        }
    }
}

```

```
func 0 (Transpose submission): hits:6066, misses:2116, evictions:2084
```

결과는 miss 횟수가 2116회로, 만점 기준인 2000회 미만을 아슬아슬하게 넘지 못했다. 해당 miss 값에 대해 나쁘지 않은 점수가 나왔고 lab을 마무리할 시간이 부족했기 때문에, 해당 코드로 61x67을 마무리하기로 했다.

driver.py를 통해 계산한 점수 결과이다.

Points (s,E,b)	Your simulator			Reference simulator			traces/yi2.trace
	Hits	Misses	Evicts	Hits	Misses	Evicts	
3 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
3 (4,2,4)	4	5	2	4	5	2	traces/yi.trace
3 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
3 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
3 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
3 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
3 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
6 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace
	27						

Part B: Testing transpose function

Running ./test-trans -M 32 -N 32

Running ./test-trans -M 64 -N 64

Running ./test-trans -M 61 -N 67

Cache Lab summary:

	Points	Max pts	Misses
Csim correctness	27.0	27	
Trans perf 32x32	8.0	8	287
Trans perf 64x64	8.0	8	1179
Trans perf 61x67	8.8	10	2115
Total points	51.8	53	